Mach Learn
DOI 10.1007/s10994-015-5488-x

Efficient inference and learning in a large knowledge base

Reasoning with extracted information using a locally groundable first-order probabilistic logic

William Yang Wang[1] · Kathryn Mazaitis[1] · Ni Lao[2] · William W. Cohen[1]

Received: 10 January 2014 / Accepted: 4 March 2015

Abstract One important challenge for probabilistic logics is reasoning with very large knowledge bases (KBs) of imperfect information, such as those produced by modern web-scale information extraction systems. One scalability problem shared by many probabilistic logics is that answering queries involves "grounding" the query—i.e., mapping it to a propositional representation—and the size of a "grounding" grows with database size. To address this bottleneck, we present a first-order probabilistic language called ProPPR in which approximate "local groundings" can be constructed in time *independent* of database size. Technically, ProPPR is an extension to *stochastic logic programs* that is biased towards short derivations; it is also closely related to an earlier relational learning algorithm called the *path ranking algorithm*. We show that the problem of constructing proofs for this logic is related to computation of *personalized PageRank* on a linearized version of the proof space, and based on this connection, we develop a provably-correct approximate grounding scheme, based on the PageRank–Nibble algorithm. Building on this, we develop a fast and easily-parallelized weight-learning algorithm for ProPPR. In our experiments, we show that learning for ProPPR is orders of magnitude faster than learning for Markov logic networks; that allowing mutual recursion (joint learning) in KB inference leads to improvements in performance; and that

Editors: Gerson Zaverucha and Vítor Santos Costa.

✉ William Yang Wang
yww@cs.cmu.edu

Kathryn Mazaitis
krivard@cs.cmu.edu

Ni Lao
nlao@google.com

William W. Cohen
wcohen@cs.cmu.edu

[1] School of Computer Science, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213, USA

[2] Google Inc., 1600 Amphitheatre Parkway, Mountain View, CA 94043, USA

Published online: 04 April 2015

ProPPR can learn weights for a mutually recursive program with hundreds of clauses defining scores of interrelated predicates over a KB containing one million entities.

Keywords Probabilistic logic · Personalized PageRank · Scalable learning

1 Introduction

Probabilistic logics are useful for many important tasks (Lowd and Domingos 2007; Fuhr 1995; Poon and Domingos 2007, 2008); in particular, such logics would seem to be well-suited for inference with the "noisy" facts that are extracted by automated systems from unstructured web data. While some positive results have been obtained for this problem (Cohen 2000), most probabilistic first-order logics are not efficient enough to be used for inference on the very large broad-coverage KBs that modern information extraction systems produce (Suchanek et al. 2007; Carlson et al. 2010). One key problem is that queries are typically answered by "grounding" the query—i.e., mapping it to a propositional representation, and then performing propositional inference—and for many logics, the size of the "grounding" can be extremely large. For instance, in probabilistic Datalog (Fuhr 1995), a query is converted to a structure called an "event expression", which summarizes all possible proofs for the query against a database; in ProbLog (De Raedt et al. 2007) and MarkoViews (Jha and Suciu 2012), similar structures are created, encoded more compactly with binary decision diagrams (BDDs); in probabilistic similarity logic (PSL) (Brocheler et al. 2010), an intentional probabilistic program, together with a database, is converted to constraints for a convex optimization problem; and in Markov Logic Networks (MLNs) (Richardson and Domingos 2006), queries are converted to a propositional Markov network. In all of these cases, the result of this "grounding" process can be large.

As a concrete illustration of the "grounding" process, Fig. 1 shows a very simple MLN and its grounding over a universe of two web pages a and b. Here, the grounding is query-independent. In MLNs, the result of the grounding is a Markov network which contains one node for every atom in the Herbrand base of the program—i.e., the number of nodes is $O(n^k)$ where k is the maximal arity of a predicate and n the number of database constants. However, even a grounding size that is only linear in the number of facts in the database, $|DB|$,

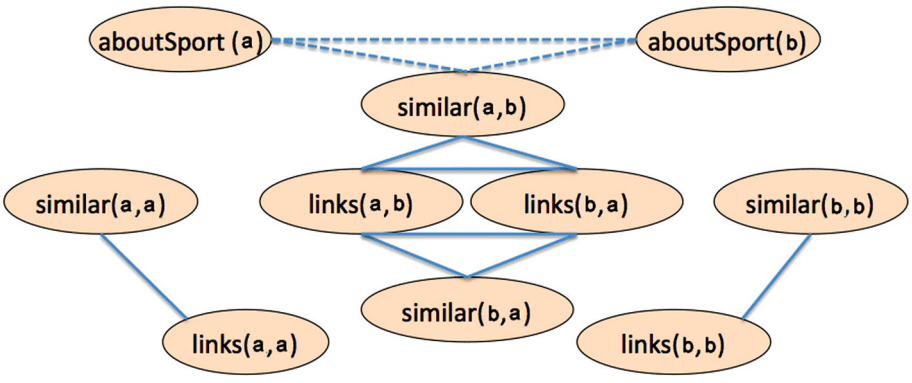


Fig. 1 A Markov logic network program and its grounding relative to two constants a, b (*dotted lines* are clique potentials associated with rule R2, *solid lines* with rule R1)

would be impractically large for inference on real-world problems. Superficially, it would seem that groundings must inherently be $o(|DB|)$ for some programs: in the example, for instance, the probability of *aboutSport(x)* must depend to some extent on the entire hyperlink graph, if it is fully connected. However, it also seems intuitive that if we are interested in inferring information about a specific page—say, the probability of *aboutSport(d1)*—then the parts of the network only distantly connected to *d1* are likely to have a small influence. This suggests that an *approximate* grounding strategy might be feasible, in which a query such as *aboutSport(d1)* would be grounded by constructing a small subgraph of the full network, followed by inference on this small "locally grounded" subgraph. As another example, consider learning from a set of queries Q with their desired truth values. Learning might proceed by locally-grounding every query goal, allowing learning to also take less than $O(|DB|)$ time.

In this paper, we first present a first-order probabilistic language which is well-suited to such approximate "local grounding". We describe an extension to *stochastic logic programs* (SLP) (Muggleton 1996; Cussens 2001) that is biased towards short derivations, and show that this is related to *personalized PageRank* (PPR) (Page et al. 1998; Chakrabarti 2007) on a linearized version of the proof space. Based on the connection to PPR, we develop a provably-correct approximate inference scheme, and an associated proveably-correct approximate grounding scheme: specifically, we show that it is possible to prove a query, or to build a graph which contains the information necessary for weight-learning, in time $O(\frac{1}{\alpha\varepsilon})$, where α is a reset parameter associated with the bias towards short derivations, and ε is the worst-case approximation error across all intermediate stages of the proof. This means that both inference and learning can be approximated in time *independent of the size of the underlying database*—a surprising and important result, which leads to a very scalable inference algorithm. We show that ProPPR is efficient enough for inference tasks on large, noisy KBs.

The ability to locally ground queries has another important consequence: it is possible to *decompose* the problem of weight-learning to a number of moderate-size subtasks—in fact, tasks of size $O(\frac{1}{\alpha\varepsilon})$ or less—which are weakly coupled. Based on this we outline a parallelization scheme, which in our current implementation provides an order-of-magnitude speedup in learning time on a multi-processor machine.

This article extends prior work (Wang et al. 2013) in the following aspects. First, the focus of this article is on inference on a noisy KB, and we comprehensively show the challenges on the inference problems on large KBs, how one can apply our proprosed locally grounding theory to improve the state-of-the-art in statistical relational learning. Second, we provide numerous new experiments on KB inference, including varying the size of the graph, comparing to MLNs, and varying the size of the theory. We demonstrate that the ProPPR inference algorithm can scale to handle million-entity datasets with several complex theories (non-recursive, PRA non-recursive, and PRA recursive). Third, we provide additional background on our approach, discussing in detail the connections to prior work on stochastic logic programs and path finding.

In the following sections, we first introduce the theoretical foundations and background of our formalism. We then define the semantics of ProPPR, and its core inference and learning procedures. We next focus on a large inference problem, and show how ProPPR can be used in a statistical relational learning task. We then present experimental results on inference in a large KB of facts extracted from the web (Lao et al. 2011). After this section, we describe our results on additional benchmark inference tasks. We finally discuss related work and conclude.

Table 1 A simple program in ProPPR. See text for explanation

```
about(X,Z):- handLabeled(X,Z)                    # base
about(X,Z):- sim(X,Y),about(Y,Z)                 # prop
sim(X,Y):- link(X,Y)                             # sim,link
sim(X,Y):-
   hasWord(X,W),hasWord(Y,W),
   linkedBy(X,Y,W)                               # sim,word
linkedBy(X,Y,W):- true                           # by(W)
```

2 Background

In this section, we introduce the necessary background that our approach builds on: logic program inference as graph search, an approximate Personalized PageRank algorithm, and stochastic logic programs.

2.1 Logic program inference as graph search

To begin with, we first show how inference in logic programs can be formulated as search over a graph. We assume some familiarity with logic programming and will use notations from Lloyd (1987). Let LP be a program which contains a set of definite clauses $c_1, \ldots, c_n$, and consider a conjunctive query Q over the predicates appearing in LP. A traditional Prolog interpreter can be viewed as having the following actions. First, construct a "root vertex" v_0, which is a pair (Q, Q), and add it to an otherwise-empty graph $G'_{Q,LP}$. (For brevity, we drop the subscripts of G' where possible.) Then recursively add to G' new vertices and edges as follows: if u is a vertex of the form $(Q, (R_1, \ldots, R_k))$, and c is a clause in LP of the form $R' \leftarrow S'_1, \ldots, S'_\ell$, and R_1 and R' have a most general unifier $\theta = mgu(R_1, R')$, then add to G' a new edge $u \rightarrow v$ where $v = (Q\theta, (S'_1, \ldots, S'_\ell, R_2, \ldots, R_k)\theta)$.[1] Let us call $Q\theta$ the *transformed query* and $(S'_1, \ldots, S'_\ell, R_2, \ldots, R_k)\theta$ the *associated subgoal list*. Empty subgoal lists correspond to solutions, and if a subgoal list is empty, we will denote it by $\square$.

The graph G' is often large or infinite so it is not constructed explicitly. Instead Prolog performs a depth-first search on G' to find the first *solution vertex* v—i.e., a vertex with an empty subgoal list—and if one is found, returns the transformed query from v as an answer to Q.

Table 1 and Fig. 2 show a simple Prolog program and a proof graph for it. (Ignore for now the annotation after the hash marks, and edge labels on the graphs, which will be introduced below.) For conciseness, in Fig. 2 only the subgoals $R_1, \ldots, R_k$ are shown in each node $u = (Q, (R_1, \ldots, R_k))$. Given the query $Q = about(a,Z)$, Prolog's depth-first search would return $Q = about(a,fashion)$. Note that in this proof formulation, the nodes are *conjunctions* of literals, and the structure is, in general, a directed graph, rather than a tree. Also note that the proof is encoded as a graph, not a hypergraph, even if the predicates in the LP are not binary: the edges represent a step in the proof that reduces one conjunction to another, not a binary relation between entities.

[1] Here $Q\theta$ denotes the result of applying the substitution θ to Q; for instance, if $Q = about(a, Z)$ and $\theta = \{Z = fashion\}$, then $Q\theta$ is $about(a, fashion)$.

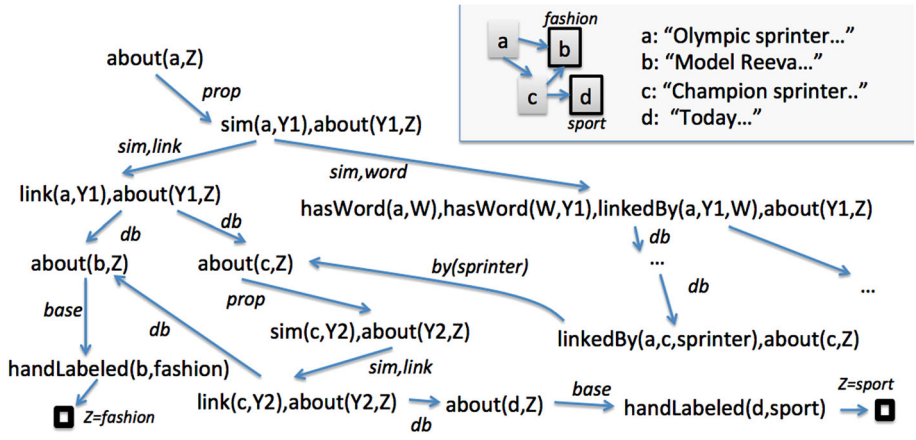


Fig. 2 A partial proof graph for the query *about(a,Z)*. The *upper right* shows the link structure between documents a, b, c, and d, and some of the words in the documents. Restart links are not shown

Table 2 The PageRank–Nibble algorithm for computing the Personalized PageRank vector in a graph where $\Pr(v|u)$ is the transition probability for reaching v from u, α' is a lower-bound on $\Pr(v_0|u)$ for any node u to be added to the graph $\hat{G}$, and ε is the desired degree of approximation

define PageRank–Nibble $(v_0, \alpha', \varepsilon)$:
  let $\mathbf{p} = \mathbf{r} = \mathbf{0}$, let $\mathbf{r}[v_0] = 1$, and let $\hat{G} = \emptyset$
  while $\exists u : \mathbf{r}(u)/|N(u)| > \varepsilon$ do:
    push(u)
  return $\mathbf{p}$
end

define push(u, α'):
  comment: this function modifies $\mathbf{p}$, $\mathbf{r}$, *and* $\hat{G}$
  $\mathbf{p}[u] = \mathbf{p}[u] + \alpha' \cdot \mathbf{r}[u]$
  $\mathbf{r}[u] = \mathbf{r}[u] \cdot (1 - \alpha')$
  for $v \in N(u)$:
    add the edge $(u, v, \phi_{u \to v})$ to $\hat{G}$
    if $v = v_0$ then
      $\mathbf{r}[v] = \mathbf{r}[v] + \Pr(v|u)\mathbf{r}[u]$
    else
      $\mathbf{r}[v] = \mathbf{r}[v] + (\Pr(v|u) - \alpha')\mathbf{r}[u]$
  endfor
end

2.2 Personalized PageRank

Personalized PageRank (Page et al. 1998) is a crucial algorithm for inference in large graphs. In this subsection, we introduce an approximate personalized PageRank method called PageRank–Nibble (Andersen et al. 2006, 2008). The outline of this algorithm is shown in Table 2. This method has been used for the problem of *local partitioning*: in local partitioning, the goal is to find a small, low-conductance[2] component $\hat{G}$ of a large graph G that contains a given node v. In the next section, we will show that this algorithm can also be used to implement a probabilistic Selective Linear Definite (SLD) resolution with bounded computation.

PageRank–Nibble maintains two vectors: $\mathbf{p}$, an approximation to the personalized PageRank vector[3] associated with node v_0, and $\mathbf{r}$, a vector of "residual errors" in $\mathbf{p}$. Initially, $\mathbf{p} = \emptyset$

[2] For small subgraphs G_S, *conductance* of G_S is the ratio of the weight of all edges exiting G_S to the weight of all edges incident on a node in G_S.

[3] In the probabilistic view of personalized PageRank, the ith entry in the vector $\mathbf{p}$ is the probability of reaching the ith node v_i from the start node v_0.

and $\mathbf{r} = \{v_0\}$. The algorithm repeatedly picks a node u with a large residual error $\mathbf{r}[u]$, and reduces this error by distributing a fraction α' of it to $\mathbf{p}[u]$, and the remaining fraction back to $\mathbf{r}[u]$ and $\mathbf{r}[v_1], \ldots, \mathbf{r}[v_n]$, where the v_i's are the neighbors of u. The order in which nodes u are picked does not matter for the analysis. In our implementation, we follow Prolog's usual depth-first search as much as possible.

Although the result stated in Andersen et al holds only for undirected graphs, it can be shown to also hold for directed graphs, as is the case here. Specifically, following their proof technique, it can be shown that after each push, $\mathbf{p} + \mathbf{r} = \mathbf{ppr}(v_0)$. It is also clear that when PageRank–Nibble terminates, then for any u, the error $\mathbf{ppr}(v_0)[u] - \mathbf{p}[u]$ is bounded by $\varepsilon|N(u)|$: hence, in any graph where the set of neighbors of u, namely $N(u)$, is bounded by a constant, a good approximation will be returned by PageRank–Nibble. PageRank–Nibble incrementally exploits the graph, and the nodes with non-zero weights are the nodes actually touched by PageRank–Nibble. Consider the subset of G' defined by these nodes (the nodes visited by the PageRank–Nibble), clearly this subgraph is also of size $o(\frac{1}{\alpha'\varepsilon})$.

To summarize, the PageRank–Nibble algorithm is an approximation to Personalized PageRank. In the next section, we will show that this algorithm can also be used to implement a probabilistic SLD resolution with bounded computation.

2.3 Stochastic logic programs

In past work on *stochastic logic programs* (SLPs) (Muggleton 1996; Cussens 2001), the randomized traversal of G' was defined by a probabilistic choice, at each node, of which clause to apply, based on a weight for each clause. Specifically, there is a weight for each clause $R_1 \leftarrow Q_1, \ldots, Q_k$ with head R_1, and these weights define a distribution on the children of node $((Q, (R_1, \ldots, R_k)))$; i.e., the clause weights defined a transition probability for the graphs G' defined above. This defines a probability distribution over vertices v, and hence a distribution over transformed queries (i.e. answers) $Q\theta$. The randomized procedure thus produces a distribution over possible answers, which can be tuned by learning to upweight desired correct answers and downweight others.

Note that each different $\square$ node corresponds to a different proof for Q, and different proofs may be associated with different substitutions θ and hence different solutions to the query Q. In Fig. 2, for instance, the leaf in the lower left corresponds to the solution $\theta = \{Z = \textit{fashion}\}$ of the query $Q = \textit{about(a,Z)}$, while the leaf in the lower right corresponds to $\{Z = \textit{sport}\}$. In the formalism of SLPs, one assigns a probabilistic score to every node in the proof graph, and then, to assign a probabilistic score to a particular solution Q', one simply sums the probabilities for all leaf nodes $(\square_i, \theta_i)$ such that $Q\theta_i = Q'$.

3 The programming with personalized PageRank (PROPPR) language

3.1 Extensions to the semantics of SLPs

We will now describe our "locally groundable" first-order probabilistic language, which we call ProPPR. Inference for ProPPR is based on a personalized PageRank process over the proof constructed by Prolog's SLD resolution theorem-prover.

3.1.1 Feature-based transition probabilities

We propose two extensions. First, we will introduce a new way of computing clause weights, which allows for a potentially richer parameterization of the traversal process. Briefly, for each use of a clause c in a proof, we associate with it a *feature vector* ϕ, which is computed based on the binding to the variables in the head of c. The feature vector ϕ for a clause is defined by the code following "#" after c. For instance, applying the clause "*sim(X,Y):- link(X,Y) # sim, link.*" always yields a vector ϕ that has unit weight on (the dimensions corresponding to) the two ground atoms *sim* and *link*, and zero weight elsewhere; likewise, applying the clause "*linkedBy(X,Y,W):- true # by(W).*" to the goal *linkedBy(a,c,sprinter)* yields a vector ϕ that has unit weight on the atom *by(sprinter)*.

To do this, the annotations after the hashmarks in the program in Table 1 are used to generate the features on the edges in Fig. 2. For example, the #*sim,link* annotation in Table 1 is instantiated by the edge in the top-left of Fig. 2, and #*by(W)* generates a feature template, which can be activated as, for example, the *by(sprinter)* edge in the middle of Fig. 2. More generally, we associate with each edge $u \rightarrow v$ in the graph a *feature vector* $\phi_{u \rightarrow v}$. This feature vector is produced indirectly, by associating with every clause $c \in LP$ a function $\Phi_c(\theta)$, which produces the vector ϕ associated with an application of c using mgu θ. As an example, if the last clause of the program in Table 1 was applied to $(Q, linkedBy(a, c, sprinter), about(c, Z))$ with mgu $\theta = \{X = a, Y = c, W = sprinter\}$ then $\Phi_c(\theta)$ would be $\{by(sprinter)\}$, if we use a set to denote a sparse vector with 0/1 weights.

In ProPPR, it is possible to have one unique feature for each clause: in fact, this is the default setting in ProPPR. However, allowing user-defined feature vectors gives us more flexibility in guiding the learning system. This allows us to decouple "feature engineering" from writing logic programs. Note that another key difference between SLPs and ProPPR is that the former associates a fixed weight/probability with each clause, whereas in ProPPR, when a feature template is involved, the weight of this clause can depend on the partial instantiation of the clause. For example, the clause "$sim(X, Y)$:- $hasWord(X, W), hasWord(Y, W), linkedBy(X, Y, W)$ # *sim,word.*" associates two weights with the feature vector for this clause. In SLP, this would be only one weight. Additionally ProPPR allows clauses like "*linkedBy(X,Y,W):- true # by(W).*" which have feature templates that allow the weight of a clause to depend on the constants used in the proof—e.g., in Table 1, depending on the actual word, the second $sim(X, Y)$ could get different weights: if W = "*champion*" the second $sim(X, Y)$ clause gets a weight of 0.9, while for W = "*the*", the weight could be 0.2. This would not be possible in SLPs, except by including a separate clause for each instantiation of W.[4] Note that *db* is a special feature indicating that a database predicate was used.

This feature vector is computed during theorem-proving, and used to annotate the edge $u \rightarrow v$ in G' created by applying c with mgu θ. Finally, an edge $u \rightarrow v$ will be traversed with probability $\Pr(v|u) \propto f(\mathbf{w}, \phi_{u \rightarrow v})$ where $\mathbf{w}$ is a parameter vector and where $f(\mathbf{w}, \phi)$ is a weighting function. (Here we use $f(\mathbf{w}, \phi) = \exp(\mathbf{w} \cdot \phi)$, but any differentiable function would be possible.) This weighting function now determines the probability of a transition, in theorem-proving, from u to v: specifically, $\Pr_{\mathbf{w}}(v|u) \propto f(\mathbf{w}, \phi_{u \rightarrow v})$. Weights in $\mathbf{w}$ default to 1.0, and learning consists of tuning these weights.

[4] We thank an anonymous reviewer for pointing out this example.

3.1.2 Restart links and solution self-links

Our second and more fundamental extension is to add additional edges in G'. Specifically we add edges (a) from every solution vertex to itself; and (b) also add an edge from every vertex to the start vertex v_0. We will call this augmented graph $G_{Q,LP}$ below (or just G if the subscripts are clear from context). The first type of link, the self-loops, serves to upweight solution nodes, and are a heuristic—all analytic results would hold without them. The second type of link, the restart links, make SLP's graph traversal a *personalized PageRank* (PPR) procedure, sometimes known as *random-walk-with-restart* (Tong et al. 2006). Self-loop links and restart links are annotated by additional feature vector functions $\Phi_{\text{restart}}(R)$ and $\Phi_{\text{selfloop}}(R)$, which are applied to the leftmost literal R of the subgoal list for u to annotate the edge $u \rightarrow v_0$.

The restart links, which link back to the start vertex, bias the traversal of the proof graph to upweight the results of *short proofs*. To see this, note that if the restart probability $P(v_0|u) = \alpha$ for every node u, then the probability of reaching any node at depth d is bounded by $(1-\alpha)^d$. Later we will show that theoretically these restart edges are important for constructing the approximated local groundings in time independent of the database size.

3.1.3 Summary of the extended proof graph

To summarize, if u is a node of the search graph, $u = (Q\theta, (R_1, \ldots, R_k))$, then the transitions from u, and their respective probabilities, are defined as follows, where Z is an appropriate normalizing constant:

- If $v = (Q\theta\sigma, (S'_1, \ldots, S'_\ell, R2, \ldots, R_k)\theta\sigma)$ is a state derived by applying the clause c (with mgu σ), then

$$\Pr_{\mathbf{w}}(v|u) = \frac{1}{Z} f(\mathbf{w}, \Phi_c(\theta\sigma))$$

- If $v = (Q, \square)$ is a solution, then

$$\Pr_{\mathbf{w}}(v|u) = \frac{1}{Z} f(\mathbf{w}, \Phi_{\text{selfloop}}(R_1\theta))$$

- If $v = v_0 = (Q, Q)$ is the initial state in G, then

$$\Pr_{\mathbf{w}}(v|u) = \frac{1}{Z} f(\mathbf{w}, \Phi_{\text{restart}}(R_1\theta))$$

- If v is any other node, then $\Pr(v|u) = 0$.

Finally we must specify the functions Φ_c and Φ_{restart}. For clauses in LP, the feature-vector producing function $\Phi_c(\theta)$ for a clause is specified by annotating c as follows: every clause $c = (R \leftarrow S_1, \ldots, S_k)$ is annotated with a conjunction of "feature literals" $F_1, \ldots, F_\ell$, which are written at the end of the clause after the special marker "#". The function $\Phi_c(\theta)$ then returns a vector $\phi = \{F_1\theta, \ldots, F_\ell\theta\}$, where every $F_i\theta$ must be ground: it cannot contain any free variables.

The requirement[5] that edge features $F_i\theta$ are ground is the reason for introducing the apparently unnecessary predicate *linkedBy(X,Y,W)* into the program of Table 1: adding the feature literal *by(W)* to the second clause for *sim* would result in a non-ground feature *by(W)*,

[5] The requirement that the feature literals returned by $\Phi_c(\theta)$ must be ground in θ is not strictly necessary for correctness. However, in developing ProPPR programs we noted that non-ground features were usually not what the programmer intended.

since W is a free variable when c is used. Notice also that the weights on the *by(W)* features are meaningful, even though there is only one clause in the definition of *linkedBy*, as the weight for applying this clause competes with the weight assigned to the restart edges.

It would be cumbersome to annotate every database fact, and difficult to learn weights for so many features. Thus, if c is the unit clause that corresponds to a database fact, then $\Phi_c(\theta)$ returns a default value. $\phi = \{db\}$.[6]

The function $\Phi_{\text{restart}}(R)$ depends on the functor and arity of R. If R is defined by clauses in LP, then $\Phi_{\text{restart}}(R)$ returns a unit vector $\phi = \{defRestart\}$. If R is a database predicate [e.g., *hasWord(doc1,W)*] then we follow a slightly different procedure, which is designed to ensure that the restart link has a reasonably large weight even with unit feature weights: we compute n, the number of possible bindings for R, and set $\phi[defRestart] = n \cdot \frac{\alpha}{1-\alpha}$, where α is a global parameter. This means that with unit weights, after normalization, the probability of following the restart link will be α.

Putting this all together with the standard power-iteration approach to computing personalized PageRank over a graph (Page et al. 1998), we arrive at the following inference algorithm for answering a query Q, using a weight vector $\mathbf{w}$. We let $\mathbf{W}$ be a matrix such that $\mathbf{W}[u, v] = \Pr_{\mathbf{w}}(v|u)$, and in our discussion, we use $\mathbf{ppr}(v_0)$ to denote the personalized PageRank vector for v_0. Let $N_{u+0}(u)$ be the neighbors of u with non-zero weight.

1. Let $v_0 = (Q, Q)$ be the start node of the search graph. Let G be a graph containing just v_0. Let $\mathbf{v}^0 = \{v_0\}$.
2. For $t = 1, \ldots, T$ (i.e., until convergence):

   For each u with non-zero weight in $\mathbf{v}^{t-1}$, and each $v \in N_{u+0}(u)$, add $(u, v, \Phi_{u \to v})$ to G with weight $\Pr_{\mathbf{w}}(v|u)$, and set $\mathbf{v}^t = \mathbf{W} \cdot \mathbf{v}^{t-1}$
3. At this point $\mathbf{v}^T \approx \mathbf{ppr}(v_0)$. Let S be the set of nodes $(Q\theta, \square)$ that have empty subgoal lists and non-zero weight in $\mathbf{v}^T$, and let $Z = \sum_{u \in S} \mathbf{v}^T[u]$. The final probability for the literal $L = Q\theta$ is found by extracting these solution nodes S, and renormalizing:

$$\Pr_{\mathbf{w}}(L) \equiv \frac{1}{Z} \mathbf{v}^T[(L, \square)]$$

where $\mathbf{v}^T[(L, \square)]$ is the unnormalized probability of L reaching this particular solution node. For example, given the query $Q = about(a,Z)$ and the program of Table 1, this procedure would assign a non-zero probability to the literals *about(a,sport)* and *about(a,fashion)*, concurrently building the graph of Fig. 2.

As a further illustration of the sorts of ProPPR programs that are possible, some small sample programs are shown in Table 3. Clauses c_1 and c_2 are, together, a bag-of-words classifier: each proof of *predictedClass(D,Y)* adds some evidence for document D having class Y, with the weight of this evidence depending on the weight given to c_2's use in establishing *related(w,y)*, where w is a specific word in D and y is a possible class label. In turn, c_2's weight depends on the weight assigned to the $r(w, y)$ feature by $\mathbf{w}$, relative to the weight of the restart link.[7] Adding c_3 and c_4 to this program implements label propagation,[8] and adding c_5 and c_6 implements a sequential classifier. These examples show that ProPPR allows many useful heuristics to be encoded as programs.

[6] If a non-database clause c has no annotation, then the default vector is $\phi = \{id(c)\}$, where c is an identifier for the clause c.

[7] The existence of the restart link thus has another important role in this program, as it avoids a sort of "label bias problem" in which local decisions are difficult to adjust.

[8] Note that we use the *inDoc* predicate for efficiency purposes; in ProPPR, like Prolog, predicates are indexed by their first argument.

Table 3 Some more sample ProPPR programs

c_1: predictedClass(Doc,Y):-
possibleClass(Y),
hasWord(Doc,W),
related(W,Y) # c1.
c_2: related(W,Y):- true,
relatedFeature(W,Y)

Database predicates:
hasWord(D,W): doc D contains word W
inDoc(W,D): doc D contains word W
previous(D1,D2): doc D2 precedes D1
possibleClass(Y): Y is a class label

c_3: predictedClass(Doc,Y):-
similar(Doc,OtherDoc),
predictedClass(OtherDoc,Y) # c3.
c_4 : similar(Doc1,Doc2):-
hasWord(Doc1,W),
inDoc(W,Doc2) # c4.

c_5 : predictedClass(Doc,Y):-
previous(Doc,OtherDoc),
predictedClass(OtherDoc,OtherY),
transition(OtherY,Y) # c5.
c_6: transition(Y1,Y2):- true,
transitionFeature(Y1,Y2)

$LP = \{c_1, c_2\}$ is a bag-of-words classifier (see text). $LP = \{c_1, c_2, c_3, c_4\}$ is a recursive label-propagation scheme, in which predicted labels for one document are assigned to similar documents, with similarity being an (untrained) cosine distance-like measure. $LP = \{c_1, c_2, c_5, c_6\}$ is a sequential classifier for document sequences

3.1.4 Discussion

Thus far, we have introduced a language quite similar to SLPs. The power-iteration PPR computation outlined above corresponds to a depth-bounded breadth-first search procedure, and the main extension of ProPPR, relative to SLPs, is the ability to label a clause application with a feature vector, instead of the clause's identifier. Below, however, we will discuss a much faster approximate grounding strategy, which leads to a novel proof strategy, and a parallelizable weight-learning method.

3.2 Locally grounding a query

Note that the procedure for computing PPR on the proof graph both performs inference by computing a distribution over literals $Q\theta$ and "grounds" the query, by constructing a graph G. ProPPR inference for this query can be re-done efficiently,[9] by running an ordinary PPR process on G. Unfortunately, the grounding G can be very large: it does not always include the entire database, but if T is the number of iterations until convergence for the sample program of Table 1 on the query $Q = about(d,Y)$, G will include a node for every page within T hyperlinks of d. To address this problem, we can use a proof procedure based on PageRank–Nibble, the algorithm described in Sect. 2.2.

The PageRank–Nibble–Prove algorithm is shown in Table 4, which calls the approximate Personalized PageRank algorithm in Table 2. Relative to PageRank–Nibble, the main differences are the use of a lower-bound on α rather than a fixed restart weight and the construction of the graph $\hat{G}$. Putting them together, we have the following efficiency bound:

Theorem 1 (Andersen, Chung, Lang) *Let u_i be the ith node pushed by PageRank–Nibble–Prove. Then, for the sum of all neighbors of u_i, we have $\sum_i |N(u_i)| < \frac{1}{\alpha'\varepsilon}$.*

This can be proved by noting that initially $||\mathbf{r}||_1 = 1$, and also that $||\mathbf{r}||_1$ decreases by at least $\alpha'\varepsilon|N(u_i)|$ on the ith push. As a direct consequence we have the following:

[9] This is useful for faster weight learning, which will be explained in the next subsection.

Table 4 The PageRank–Nibble–Prove algorithm for inference in ProPPR. α' is a lower-bound on $\Pr(v_0|u)$ for any node u to be added to the graph $\hat{G}$, and ε is the desired degree of approximation of the **ppr** vector

define PageRank–Nibble–Prove(Q):
  let $\mathbf{v}$ =PageRank–Nibble($(Q, Q), \alpha', \varepsilon$)
  let $S = \{u : \mathbf{p}[u] > \varepsilon$ and $u = (Q\theta, \square)\}$
  let $Z = \sum_{u \in S} \mathbf{p}[u]$
  define $\Pr_{\mathbf{w}}(L) \equiv \frac{1}{Z}\mathbf{v}[(L, \square)]$
end

The nodes touched by PageRank–Nibble also define a subset $\hat{G}$ of the proof graph G

Corollary 1 *The number of edges in the graph $\hat{G}$ produced by PageRank–Nibble–Prove is no more than $\frac{1}{\alpha'\varepsilon}$.*

Note that we expand the proof tree gradually, and generate a partial proof tree $\hat{G_Q}$ during this process. Importantly, the bound holds *independent of the size of the full database of facts*. The bound also holds regardless of the size or loopiness of the full proof graph, so this inference procedure will work for recursive logic programs.[10] Note that the bound depends on $1/\alpha'$, so if $\alpha' = 0$, then this approximate PageRank procedure may not terminate.

We should emphasize a limitation of this analysis: this approximation result holds for the individual nodes in the proof tree, not the answers $Q\theta$ to a query Q. Following SLPs, the probability of an answer $Q\theta$ is the sum of the weights of all solution nodes that are associated with θ, so if an answer is associated with n solutions, the error for its probability estimate with PageRank–Nibble–Prove may be as large as $n\varepsilon$.

To summarize, we have outlined an efficient approximate proof procedure, which is closely related to personalized PageRank. As a side-effect of inference for a query Q, this procedure will create a ground graph $\hat{G}_Q$ on which personalized PageRank can be run directly, without any relatively expensive manipulation of first-order theorem-proving constructs such as clauses or logical variables. As we will see, this "locally grounded"[11] graph will be very useful in learning weights $\mathbf{w}$ to assign to the features of a ProPPR program.

3.3 Learning for ProPPR

As noted above, inference for a query Q in ProPPR is based on a personalized PageRank process over the graph associated with the SLD proof of a query goal G. More specifically, the edges $u \to v$ of the graph G are annotated with feature vectors $\phi_{u \to v}$, and from these feature vectors, weights are computed using a parameter vector $\mathbf{w}$, and finally normalized to form a probability distribution over the neighbors of u. The "grounded" version of inference is thus a personalized PageRank process over a graph with feature-vector annotated edges.

In prior work, Backstrom and Leskovec (2011) outlined a family of supervised learning procedures for this sort of annotated graph. In the simpler case of their learning procedure, an example is a triple (v_0, u, y) where v_0 is a query node, u is a node in the personalized PageRank vector $\mathbf{p}_{v_0}$ for v_0, y is a target value, and a loss $\ell(v_0, u, y)$ is incurred if $\mathbf{p}_{v_0}[u] \neq y$.

[10] For undirected graphs, it can also be shown (Andersen et al. 2006, 2008) that the subgraph $\hat{G}$ is in some sense a "useful" subset of the full proof space: for an appropriate setting of ε, if there is a low-conductance subgraph G_* of the full graph that contains v_0, then G_* will be contained in $\hat{G}$: thus if there is a subgraph G_* containing v_0 that approximates the full graph well, PageRank–Nibble will find (a supergraph of) G_*.

[11] Note that "local grounding" means constructing a proof graph, but the nodes in this graph need not to be ground terms in the logic programming sense.

In the more complex case of "learning to rank", an example is a triple (v_0, u_+, u_-) where v_0 is a query node, u_+ and u_- are nodes in the personalized PageRank vector $\mathbf{p}_{v_0}$ for v_0, and a loss is incurred unless $\mathbf{p}_{v_0}[u_+] \geq \mathbf{p}_{v_0}[u_-]$. The core of Backstrom and Leskovec's result is a method for computing the gradient of the loss on an example, given a differentiable feature-weighting function $f(\mathbf{w}, \phi)$ and a differentiable loss function ℓ. The gradient computation is broadly similar to the power-iteration method for computation of the personalized PageRank vector for v_0. Given the gradient, a number of optimization methods can be used to compute a local optimum.

Instead of directly using the above learning approach for ProPPR, we decompose the pairwise ranking loss into a standard positive-negative log loss function. The training data D is a set of triples $\{(Q^1, P^1, N^1), \ldots, (Q^m, P^m, N^m)\}$ where each Q^k is a query, $P^k = \langle Q\theta^1_+, \ldots, Q\theta^I_+ \rangle$ is a list of correct answers, and N^k is a list $\langle Q\theta^1_-, \ldots, Q\theta^J_- \rangle$ of incorrect answers. We use a log loss with L_2 regularization of the parameter weights. Hence the final function to be minimized is

$$-\left(\sum_{k=1}^{I} \log \mathbf{p}_{v_0}[u^k_+] + \sum_{k=1}^{J} \log(1 - \mathbf{p}_{v_0}[u^k_-]) \right) + \mu ||\mathbf{w}||_2^2$$

To optimize this loss, we use stochastic gradient descent (SGD), rather than the quasi-Newton method of Backstrom and Leskovic. Weights are initialized to $1.0 + \delta$, where δ is randomly drawn from $[0, 0.01]$. We set the learning rate β of SGD to be $\beta = \frac{\eta}{\text{epoch}^2}$ where epoch is the current epoch in SGD, and η, the initial learning rate, defaults to 1.0.

We implemented SGD because it is fast and has been adapted to parallel learning tasks (Zinkevich et al. 2010; Recht et al. 2011). Local grounding means that learning for ProPPR is quite well-suited to parallelization. The step of locally grounding each Q^i is "embarrassingly" parallel, as every grounding can be done independently. To parallelize the weight-learning stage, we use multiple threads, each of which computes the gradient over a single grounding $\hat{G}_{Q^i}$, and all of which access a single shared parameter vector $\mathbf{w}$. The shared parameter vector is a potential bottleneck (Zinkevich et al. 2009); while it is not a severe one on moderate-size problems, contention for the parameters does become increasingly important with many threads.

4 Inference in a noisy KB

In this section, we focus on improving the state-of-the-art in learning inference rules for a noisy KB using ProPPR. The road map is as follows:

- first, we introduce the challenges of inference in a noisy KB in the next subsection;
- second, in Sect. 4.2, we describe the background of PRA (Lao et al. 2011), which is a state-of-the-art algorithm that learns non-recursive theories of a particular type;
- finally, we then show in Sect. 4.3 how one can use ProPPR to form a recursive theory of PRA's learned rules to improve this learning scheme.

4.1 Challenges of inference in a noisy KB

A number of recent efforts in industry (Singhal 2012) and academia (Suchanek et al. 2007; Carlson et al. 2010; Hoffmann et al. 2011) have focused on automatically constructing large knowledge bases (KBs). Because automatically-constructed KBs are typically imperfect and incomplete, inference in such KBs is non-trivial.

We situate our study in the context of the Never Ending Language Learning (NELL) research project, which is an effort to develop a never-ending learning system that operates 24 h per day, for years, to continuously improve its ability to extract structured facts from the web (Carlson et al. 2010). NELL is given as input an ontology that denotes hundreds of categories (e.g., person, beverage, athlete, sport) and two-place typed relations among these categories [e.g., athletePlaysSport (Athlete, Sport)], which it must learn to extract from the web. NELL is also provided a set of 10–20 positive seed examples of each such category and relation, along with a downloaded collection of 500 million web pages from the ClueWeb2009 corpus (Callan et al. 2009) as unlabeled data, and access to 100,000 queries each day to Google's search engine. NELL uses a multi-strategy semi-supervised multi-view learning method to iteratively grow the set of extracted "beliefs".

This task is challenging for two reasons. First, the extensional knowledge inference is not only incomplete, but also noisy, since it is extracted imperfectly from the web. For example, a football team might be wrongly recognized as two separate entities, one with connections to its team members, and the other with a connection to its home stadium. Second, the sizes of inference problems are large relative to those in many other probabilistic inference tasks. Given the very large broad-coverage KBs that modern information extraction systems produce (Suchanek et al. 2007; Carlson et al. 2010), even a grounding of size that is only linear in the number of facts in the database, $|DB|$, would be impractically large for inference on real-world problems.

Past work on first-order reasoning has sought to address the first problem by learning "soft" inference procedures, which are more reliable than "hard" inference rules, and to address the second problem by learning restricted inference procedures. In the next sub-section, we will recap a recent development in solving these problems, and draw a connection to the ProPPR language.

4.2 Inference using the path ranking algorithm (PRA)

Lao et al. (2011) use the path ranking algorithm (PRA) to learn an "inference" procedure based on a weighted combination of "paths" through the KB graph. PRA is a relational learning system which generates and appropriately weights rules, which accurately infer new facts from the existing facts in the noisy knowledge base. As an illustration, PRA might learn rules such as those in Table 5, which correspond closely to Horn clauses, as shown in the table.

PRA only considers rules which correspond to "paths", or chains of binary, function-free predicates. Like ProPPR, PRA will weight some solutions to these paths more heavily than

Table 5 Example PRA rules learned from NELL, written as Prolog clauses

PRA Paths for inferring **athletePlaysSport**:
athletePlaysSport(A,S):- factAthletePlaysForTeam(A,T),factTeamPlaysSport(T,S).
PRA Paths for inferring **teamPlaysSport**:
teamPlaysSport(T,S):-
factMemberOfConference(T,C),factConferenceHasMember(C,T'),factTeamPlaysSport(T',S).
teamPlaysSport(T,S):-
factTeamHasAthlete(T,A),factAthletePlaysSport(A,S).

others: specifically, weights of the solutions to a PRA "path" are based on random-walk probabilities in the corresponding graph. For instance, the last clause of Table 5, which corresponds to the "path"

$$T \xrightarrow{\text{teamHasAthlete}} A \xrightarrow{\text{athletePlaysSport}} S$$

can be understood as follows:

1. Given a team T, construct a uniform distribution $\mathscr{A}$ of athletes such that $A \in \mathscr{A}$ is a athlete playing for team T.
2. Given $\mathscr{A}$, construct a distribution of sports $\mathscr{S}$ such that $S \in \mathscr{S}$ is played by A.

This final distribution $\mathscr{S}$ is the result: thus the path gives a weighted distribution over possible sports played by a team.

The paths produced by PRA, together with their weighting scheme, corresponds precisely to ProPPR clauses. More generally, the output of PRA corresponds roughly to a ProPPR program in a particular form—namely, the form

$$\begin{aligned} p(S,T) &\leftarrow r_{1,1}(S,X_1), r_{1,2}(X_1,X_2), \ldots, r_{1,k_1}(X_{k_1-1},T). \\ p(S,T) &\leftarrow r_{2,1}(S,X_1), r_{2,2}(X_1,X_2), \ldots, r_{2,k_2}(X_{k_2-1},T). \\ &\vdots \end{aligned}$$

where p is the binary predicate being learned, and the $r_{i,j}$'s are other predicates defined in the database. In Table 5, we emphasize that the $r_{i,j}$'s are already extensionally defined by prefixing them with the string "fact". PRA generates a very large number of such rules, and then combines them using a sparse linear weighting scheme, where the weighted solutions associated with a single "path clause" are combined with learned parameter weights to produce a final ranking over entity pairs. More formally, following the notation of Lao and Cohen (2010), we define a *relation path* P as a sequence of relations $r_1, \ldots, r_\ell$. For any relation path $P = r_1, \ldots, r_\ell$, and seed node s, a *path constrained random walk* defines a distribution h as $h_{s,P}(e) = 1$ if $e = s$, and $h_{s,p}(e) = 0$ otherwise. If P is not empty, then $P' = r_1, \ldots, r_{\ell-1}$, such that:

$$h_{s,P}(e) = \sum_{e' \in P'} h_{s,P'}(e') \cdot P(e|e'; r_\ell) \tag{1}$$

where the term $P(e|e'; r_\ell)$ is the probability of reaching node e from node e' with a one-step random walk with edge type r_ℓ; that is, it is $\frac{1}{k}$, where $k = |\{e' : r_\ell(e, e')\}|$, i.e., the number of entities e' related to e via the relation r_ℓ.

Assume we have a set of paths $P_1, \ldots, P_n$. The PRA algorithm treats each entity-pair $h_{s,P}(e)$ as a *path feature* for node e, and ranks entities using a linear weighting scheme:

$$w_1 h_{s,P_1}(e) + w_2 h_{s,P_2}(e) + \cdots + w_n h_{s,P_n}(e) \tag{2}$$

where w_i is the weight for the path P_i. PRA then learns the weights $\mathbf{w}$ by performing elastic net-like regularized maximum likelihood estimation of the following objective function:

$$\sum_i j_i(w) - \mu_1 \|\mathbf{w}\|_1 - \mu_2 \|\mathbf{w}\|_2^2 \tag{3}$$

Here μ_1 and μ_2 are regularization coefficients for elastic net regularization, and the loss function $j_i(w)$ is the per-instance objective function. The regularization on $\|\mathbf{w}\|_1$ tends to

Table 6 Example recursive Prolog rules constructed from PRA paths

Rules for inferring **athletePlaysSport**:
athletePlaysSport(A,S):- factAthletePlaysSport(A,S).
athletePlaysSport(A,S):- athletePlaysForTeam(A,T),teamPlaysSport(T,S).
Rules for inferring **teamPlaysSport**:
teamPlaysSport(T,S):- factTeamPlaysSport(T,S).
teamPlaysSport(T,S):- memberOfConference(T,C),conferenceHasMember(C,T'),teamPlaysSport(T',S).
teamPlaysSport(T,S):- teamHasAthlete(T,A),athletePlaysSport(A,S).

drive weights to zero, which allows PRA to produce a sparse classifier with relatively small number of path clauses. More details on PRA can be found elsewhere (Lao and Cohen 2010).

4.3 From non-recursive to recursive theories: joint inference for multiple relations

One important limitation of PRA is that it learns only programs in the limited form given above. In particular, PRA can not learn or even execute recursive programs, or programs with predicates of arity more than two. PRA also must learn each predicate definition completely independently.

To see why this is a limitation consider the program in Table 5, which could be learned by PRA by invoking it twice, once for the predicate *athletePlaysSport* and once for *teamPlaysSport*. We call this formulation the *non-recursive formulation* for a theory. An alternative would be to define two mutually recursive predicates, as in Table 6. We call this the *recursive formulation*. Learning weights for theories written using the recursive formulation is a *joint* learning task, since several predicates are considered together. In the next section, we ask the question: can joint learning, via weight-learning of mutually recursive programs of this sort, improve performance for a learned inference scheme for a KB?

5 Experiments in KB inference

To understand the locally groundable first-order logic in depth, we investigate ProPPR on the difficult problem of drawing reliable inferences from imperfectly extracted knowledge. In this experiment, we create training data by using NELL's KB as of iteration 713, and test, using as positive examples new facts learned by NELL in later iterations. Negative examples are created by sampling beliefs from relations that are mutually exclusive with the target relation. Throughout this section, we set the number of SGD optimization epochs to 10. Since PRA has already applied the elastic net regularizer when learning the weights of different rules, and we are working with multiple subsets with various sizes of input, μ was set to 0 in ProPPR's SGD learning in this section.

For experimental purposes, we construct a number of varying-sized versions of the KB using the following procedure. First, we construct a "knowledge graph", where the nodes are entities and the edges are the binary predicates from NELL. Then, we pick a seed entity s, and find the M entities that are ranked highest using a simple untyped random walk with restart over the full knowledge graph from seed s. Finally, we project the KB to just these M entities: i.e., we select all entities in this set, and all unary and binary relationships from the original KB that concern only these M entities.

This process leads to a well-connected knowledge base of bounded size, and by picking different seeds s, we can create multiple different knowledge bases to experiment on. In the experiments below, we used the seeds "Google", "The Beatles", and "Baseball" obtaining KBs focused on technology, music, and sports, respectively.

In this section, we mainly look at three types of rules:

- *KB non-recursive*: the simple non-recursive KB rules that do not contain PRA paths (e.g. teamPlaysSport(T,S):- factTeamPlaysSport(T,S).);
- *PRA non-recursive*: the non-recursive PRA rules (e.g. rules in Table 5);
- *PRA recursive*: the recursive formulation of PRA rules (e.g. rules in Table 6).

At the time of writing, there is no structure-learning component for ProPPR, we construct a program by taking the top-weighted k rules produced PRA for each relation, for some value of k, and then syntactically transforming them into ProPPR programs, using either the recursive or non-recursive formulation, as described in Tables 5 and 6 respectively. Again, note that the recursive formulation allows us to do joint inference on all the learned PRA rules for all relations at once.

5.1 Varying the size of the graph

To explore the scalability of the system on large tasks, we evaluated the performance of ProPPR on NELL KB subsets that have $M = 100{,}000$ and $M = 1{,}000{,}000$ entities. In these experiments, we considered only the top-weighted PRA rule for each defined predicate. On the 100K subsets, we have 234, 180, and 237 non-recursive KB rules, and 534, 430, and 540 non-recursive/recursive PRA rules in the Google, Beatles, and Baseball KBs, respectively.[12] On the 1M subsets, we have 257, 253, and 255 non-recursive KB rules, and 569, 563, and 567 non-recursive/recursive PRA rules for the three KBs. We set $\varepsilon = 0.01$ and $\alpha = 0.1$. For example, here queries are in the form of "headquarterInCity(Google,?)", where as positive/negative examples are "+headquarterInCity(Google,MountainView)", and "−headquarterInCity(Google,Pittsburgh)".

First we examine the AUC[13] of non-recursive KB rules, non-recursive PRA and recursive PRA ProPPR theories, after weight-learning, on the 100K and 1M subsets. From Table 7, we see that the recursive formulation performs better in all subsets. Performance on the 1M KBs are similar, because it turns out the largest KBs largely overlap. (This version of the NELL KB has only a a little more than one million entities involved in binary relations.) When examining the learned weights of the recursive program, we notice that the top-ranked rules are the recursive PRA rules, as we expected.

In the second experiment, we consider the training time for ProPPR, and in particular, how multithreaded SGD training affects the training time. Table 8 shows the runtime for the multithreaded SGD on the NELL 100K and 1M datasets. Learning takes less than two minute for all the data sets, even on a single processor, and multithreading reduces this to less than 20 s. Hence, although we have not observed perfect speedup (probably due to parameter-vector contention) it is clear that SGD is fast, and that parallel SGD can significantly reduce the training time for ProPPR.

[12] Note that because of the additional recursive rules, the number of rules in non-recursive/recursive case is much larger than using non-recursive only.

[13] Throughout this section, all the AUCs are areas under the ROC curve.

Table 7 Comparing the learning algorithm's AUC among non-recursive KB, non-recursive PRA, and recursive formulation of ProPPR on NELL 100K and 1M datasets

The best results are highlighted in bold

Methods	Google	Beatles	Baseball
ProPPR 100K KB non-recursive	0.699	0.679	0.694
ProPPR 100K PRA non-recursive	0.942	0.881	0.943
ProPPR 100K PRA recursive	**0.950**	**0.884**	**0.952**
ProPPR 1M KB non-recursive	0.701	0.701	0.700
ProPPR 1M PRA non-recursive	0.945	0.944	0.945
ProPPR 1M PRA recursive	**0.955**	**0.955**	**0.955**

Table 8 Runtime (seconds) for parallel SGD while training the recursive formulation of ProPPR on NELL 100K and 1M datasets

#Threads	Google	Beatles	Baseball
100K			
1	54.9	20.0	51.4
2	29.4	12.1	26.6
4	19.1	7.4	16.8
8	12.1	6.3	13.0
16	9.6	5.3	9.2
1M			
1	116.4	87.3	111.7
2	52.6	54.0	59.4
4	31.0	33.0	31.3
8	19.0	21.4	19.1
16	15.0	17.8	15.7

5.2 Comparing ProPPR and MLNs

Next we quantitatively compare ProPPR's inference time, learning time, and performance with MLNs, using the Alchemy toolkit.[14] We train with a KB with $M = 1000$ entities,[15] and test with a KB with $M = 10{,}000$. The number of non-recursive KB rules is 95, 10, and 56 respectively, and the corresponding number of non-recursive/recursive PRA rules are 230, 29, and 148. The number of training queries are 466, 520, and 130, and the number of testing queries are 3143, 2552, and 4906. We set $\varepsilon = 0.01$ and $\alpha = 0.1$. Again, we only take the top-1 PRA paths to construct ProPPR programs in this subsection.

In the first experiment, we investigate whether inference in ProPPR is sensitive to the size of graph. Using MLNs and ProPPR non-recursive KB programs trained on the 1K training subsets, we evaluate the inference time on the 10K testing subsets by varying the number of entities in the database used at evaluation time. Specifically, we use a fixed number of test queries, and increase the total number of entities in the KB by a factor of X, for various values of X. In Fig. 3, we see that ProPPR's runtime is independent of the size of the KB. In contrast, when comparing to MC-SAT, the default and most efficient inference method in MLN, we observe that inference time slows significantly when the database size grows.

[14] http://alchemy.cs.washington.edu/.

[15] We were unable to train MLNs with more than 1000 entities.

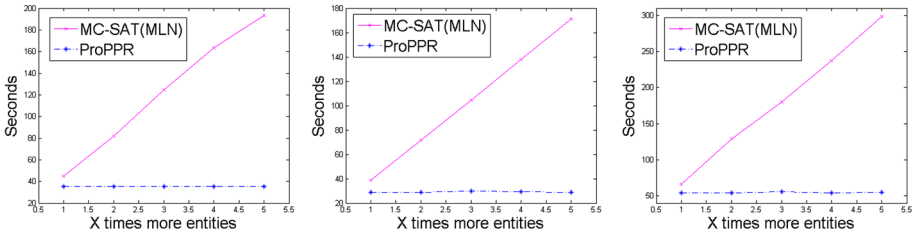


Fig. 3 Run-time for non-recursive KB inference on NELL 10K subsets the using ProPPR (with a single thread) as a function of increasing the total entities by X times in the database. Total test queries are fixed in each subdomain. *Left* the Google 10K dataset; middle, the Beatles 10K dataset; *right* the Baseball 10K dataset

Table 9 Comparing the learning algorithm's runtime between ProPPR and MLNs on the NELL 1K subsets

Method	Google	Beatles	Baseball
ProPPR SGD KB non-recursive	2.6	2.3	1.5
MLN conjugate gradident	8604.3	1177.4	5172.9
MLN voted perceptron	8581.4	967.3	4194.5
ProPPR SGD PRA non-recursive	2.6	3.4	1.7
ProPPR SGD PRA recursive	4.7	3.5	2.1

Table 10 Comparing the learning algorithm's AUC between recursive formulation of ProPPR and MLNs

The best results are highlighted in bold

Methods	Google	Beatles	Baseball
ProPPR SGD KB non-recursive	0.568	0.510	0.652
MLN conjugate gradident	0.716	0.544	0.645
MLN voted perceptron	0.826	0.573	0.672
ProPPR SGD PRA non-recursive	0.894	**0.922**	0.930
ProPPR SGD PRA recursive	**0.899**	0.899	**0.935**

In the second experiment, we compare ProPPR's SGD training method with MLNs most efficient discriminative learning methods: voted perceptron and conjugate gradient (Lowd and Domingos 2007). To do this, we fixed the number of iterations of discriminative training in MLN to 10, and also fixed the number of SGD passes in ProPPR to 10. In Table 9, we show the runtime of various approaches on the three NELL subdomains. When running on the non-recursive KB theory, ProPPR has averages 1–2 s runtime across all domains, whereas training MLNs takes hours. When training on the non-recursive/recursive PRA theories, ProPPR is still efficient.[16]

We now examine the accuracy of ProPPR, in particular, the recursive formulation, and compare with MLN's popular discriminative learning methods: voted perceptron and conjugate gradient. In Table 10, we see that MLNs outperform ProPPR's using the non-recursive formulation. However, ProPPR's recursive formulation outperforms all other methods, and shows the benefits of joint inference with recursive theories.

We should emphasize that the use of AUC means that we are evaluating only the *ranking* of the possible answers to a query; in other words, we are not measuring the quality of the actual probability scores produced by ProPPR, only the relative scores for a particular query.

[16] We were unable to train MLNs with non-recursive or recursive PRA rules.

Table 11 AUCs for using top-k PRA paths for recursive formulation of ProPPR on NELL 100K and 1M datasets

Methods	Google	Beatles	Baseball
ProPPR 100K top-1 recursive	0.950	0.884	**0.952**
ProPPR 100K top-2 recursive	0.954	0.916	0.950
ProPPR 100K top-3 recursive	**0.959**	**0.953**	**0.952**
ProPPR 1M top-1 recursive	0.955	0.955	0.955
ProPPR 1M top-2 recursive	0.961	0.960	0.960
ProPPR 1M top-3 recursive	**0.964**	**0.964**	**0.964**

The best results are highlighted in bold

ProPPR's random-walk scores tend to be very small for all potential answers, and are not well-suited to estimating probabilities in its current implementation.

5.3 Varying the size of the theory

So far, we have observed improved performance using the recursive theories of ProPPR, constructed from top $k = 1$ PRA paths for each relation. Here we consider further increasing the size of the ProPPR program by including more PRA rules in the theory. In particular, we also extract the top-2 and top-3 PRA paths, limiting ourselves to rules with positive weights. On the 100K datasets, this increased the number of clauses in the recursive theories to 759, 624, and 765 in the Google, Beatles, and Baseball subdomains in the top-2 condition, and to 972, 806, and 983 in the top-3 condition. On the 1M datasets, we have now 801, 794, and 799 clauses in the top-2 case, and 1026, 1018, and 1024 in the top-3 setup. From Table 11, we observe that using more PRA paths improves performance on all three subdomains.

6 Experiments on other tasks

As a further test of generality, we now present results using ProPPR on two other, smaller tasks. Our first sample task is an entity resolution task previously studied as a test case for MLNs (Singla and Domingos 2006a). The program we use in the experiments is shown in Table 12: it is approximately the same as the MLN(B+T) approach from Singla and Domingos.[17] To evaluate accuracy, we use the Cora dataset, a collection of 1295 bibliography citations that refer to 132 distinct papers. We set the regularization coefficient μ to 0.001 and the number of epochs to 5.

Our second task is a bag-of-words classification task, which was previously studied as a test case for both ProbLog (Gutmann et al. 2010) and MLNs (Lowd and Domingos 2007). In this experiment, we use the following ProPPR program:

```
class(X,Y):- has(X,W), isLabel(Y), related(W,Y).
related(W,Y):- true # w(W,Y).
```

which is a bag-of-words classifier that is approximately[18] the same as the ones used in prior work (Gutmann et al. 2010; Lowd and Domingos 2007). The dataset we use is the WebKb

[17] The principle difference is that we do not include tests on the absence of words in a field in our clauses, and we drop the non-horn clauses from their program.

[18] Note that we do not use the negation rule and the link rule from Lowd and Domingos.

Table 12 ProPPR program used for entity resolution

```
samebib(BC1,BC2):-
        author(BC1,A1),sameauthor(A1,A2),authorinverse(A2,BC2)          # author
samebib(BC1,BC2):-
        title(BC1,A1),sametitle(A1,A2),titleinverse(A2,BC2)             # title
samebib(BC1,BC2):-
        venue(BC1,A1),samevenue(A1,A2),venueinverse(A2,BC2)             # venue
samebib(BC1,BC2):-
        samebib(BC1,BC3),samebib(BC3,BC2)                               # tcbib
sameauthor(A1,A2):-
        haswordauthor(A1,W),haswordauthorinverse(W,A2),keyauthorword(W) # authorword
sameauthor(A1,A2):-
        sameauthor(A1,A3),sameauthor(A3,A2)                             # tcauthor
sametitle(A1,A2):-
        haswordtitle(A1,W),haswordtitleinverse(W,A2),keytitleword(W)    # titleword
sametitle(A1,A2):-
        sametitle(A1,A3),sametitle(A3,A2)                               # tctitle
samevenue(A1,A2):-
        haswordvenue(A1,W),haswordvenueinverse(W,A2),keyvenueword(W)    # venueword
samevenue(A1,A2):-
        samevenue(A1,A3),samevenue(A3,A2)                               # tcvenue
keyauthorword(W):- true                                                 # authorWord(W)
keytitleword(W):- true                                                  # titleWord(W)
keyvenueword(W):- true                                                  # venueWord(W)
```

Table 13 Performance of the approximate PageRank–Nibble–Prove method on the Cora dataset, compared to the grounding by running personalized PageRank to convergence (power iteration)

ε	MAP	Time (s)
0.0001	0.30	28
0.00005	0.40	39
0.00002	0.53	75
0.00001	0.54	116
0.000005	0.54	216
Power iteration	0.54	819

In all cases $\alpha = 0.1$

dataset, which includes a set of web pages from four computer science departments (Cornell, Wisconsin, Washington, and Texas). Each web page has one or multiple labels: *course, department, faculty, person, research project, staff, and student*. The task is to classify the given URL into the above categories. This dataset has a total of 4165 web pages. Using our ProPPR program, we learn a separate weight for each word for each label.

6.1 Efficiency

For these smaller problems, we can evaluate the cost of the PageRank–Nibble–Prove inference/grounding technique relative to a power-iteration based prover. Table 13 shows the time

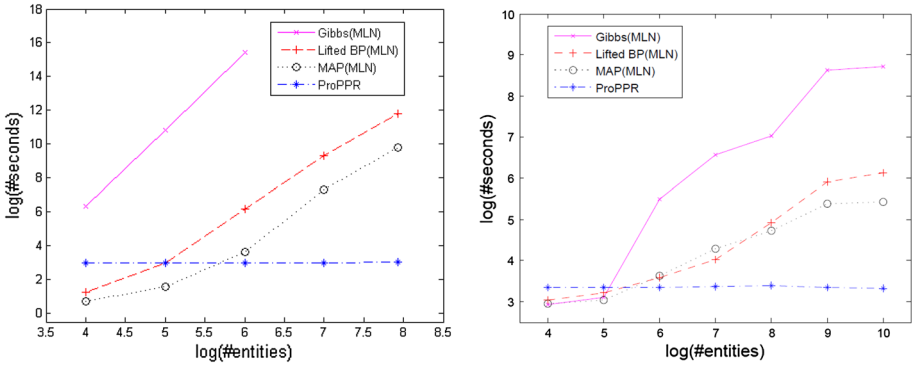
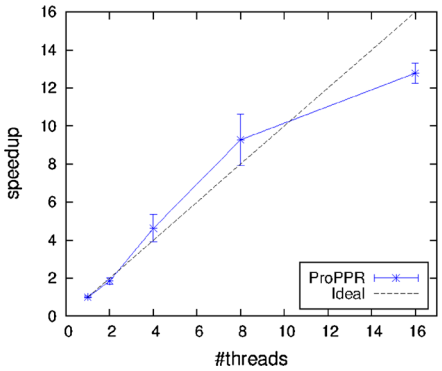


Fig. 4 Run-time for inference when using ProPPR (with a single thread) as a function of the number of entities in the database. The base of the log is 2. *Left* the Cora dataset; *right* the WebKB dataset



	Co.	Wi.	Wa.	Te.	Avg.
1	1190.4	504.0	1085.9	1036.4	954.2
2	594.9	274.5	565.7	572.5	501.9
4	380.6	141.8	404.2	396.6	330.8
8	249.4	94.5	170.2	231.5	186.4
16	137.8	69.6	129.6	141.4	119.6

Fig. 5 Performance of the parallel SGD method on the Cora and WebKB datasets. The x axis is the number of threads on a multicore machine, and the y axis is the speedup factor over a single-threaded implementation. *Co.* test on the Cornell subset, *Wi.* test on the Wisconsin subset, *Wa.* test on the Washington subset, *Te.* test on the Texas subset

required for inference with uniform weights for a set of 52 randomly chosen entity-resolution tasks from the Cora dataset, using a Python implementation of the theorem-prover. We report the time in seconds for all 52 tasks, as well as the mean average precision (MAP) of the scoring for each query. It is clear that PageRank–Nibble–Prove offers a substantial speedup on these problems with little loss in accuracy: on these problems, the same level of accuracy is achieved in less than a tenth of the time.

While the speedup in inference time is desirable, the more important advantages of the local grounding approach are that (1) grounding time, and hence inference, need not grow with the database size and (2) learning can be performed in parallel, by using multiple threads for parallel computations of gradients in SGD. Figure 4 illustrates the first of these points: the scalability of the PageRank–Nibble–Prove method as the database size increases. For comparison, we also show the inference time for MLNs with three inference methods: Gibbs refers to Gibbs sampling, Lifted BP is the lifted belief propagation method, and MAP is the maximum a posteriori inference approach. In each case the performance task is inference over 16 test queries.

Table 14 AUC results on Cora citation-matching

	Cites	Authors	Venues	Titles
MLN (Fig. 1)	0.513	0.532	0.602	0.544
MLN (S&D)	0.520	0.573	0.627	0.629
ProPPR (**w** = 1)	0.680	0.836	0.860	**0.908**
ProPPR	**0.800**	**0.840**	**0.869**	0.900

The best results are highlighted in bold

Note that ProPPR's runtime is constant, independent of the database size: it takes essentially the same time for $2^8 = 256$ entities as for $2^4 = 16$. In contrast, lifted belief propagation is up to 1000 times slower on the larger database.

Figure 5 explores the speedup in learning due to multi-threading. The weight-learning is using a Java implementation of the algorithm which runs over ground graphs. For Cora, the speedup is nearly optimal, even with 16 threads running concurrently. For WebKB, while learning time averages about 950 s with a single thread, it can be reduced to about 120 s if 16 threads are used. For comparison, Lowd and Domingos report that around 10,000 s were needed to obtain the best results for MLNs.

6.2 Effectiveness of learning

We finally consider the effectiveness of weight learning. For Cora, we train on the first four sections of the Cora dataset, and report results on the fifth. Table 14 shows AUC on the test set used by Singla and Domingos for several methods. The line for MLN (Fig. 1) shows results obtained by an MLN version of the program of Fig. 1. The line MLN (S&D) shows analogous results for the best-performing MLN from Singla and Domingos (2006a). Compared to these methods, ProPPR does quite well even before training (with unit feature weights, **w** = 1); the improvement here is likely due to the ProPPR's bias towards short proofs, and the tendency of the PPR method to put more weight on shared words that are rare, and hence have lower fanout in the graph walk. Training ProPPR improves performance on three of the four tasks, and gives the most improvement on citation-matching, the most complex task.

The results in Table 14 all use the same data and evaluation procedure, and the MLNs were trained with the state-of-the-art Alchemy system using the recommended commands for this data, which is distributed with Alchemy.[19] However, we should note that the MLN results reproduced here are not identical to previous-reported ones (Singla and Domingos 2006a). Singla and Domingos used a number of complex heuristics that are difficult to reproduce—e.g., one of these was combining MLNs with a heuristic, TFIDF-based matching procedure based on canopies (McCallum et al. 2000). While the trained ProPPR model outperforms the reproduced MLN model in all prediction tasks, it outperforms the reported results from Singla and Domingos only on *venue*, and does less well than the reported results on *citation* and *author*.[20]

On the Webkb dataset, we use the usual cross-validation method (Lowd and Domingos 2007; Gutmann et al. 2010): in each fold, for the four universities, we train on three, and report result on the fourth. In Table 15, we show the detailed AUC results of each fold, as well as the averaged results. If we do not perform weight learning, the averaged result is equivalent to a random baseline. As reported by Gutmann et al. the ProbLog approach

[19] http://alchemy.cs.washington.edu.

[20] Performance on *title* matching is not reported by Singla and Domingos.

obtains an AUC of 0.606 on the dataset (Gutmann et al. 2010), and as reported by Lowd and Domingos, the results for voted perceptron algorithm (MLN VP, AUC $\approx$ 0.605) and the contrastive divergence algorithm (MLN CD, AUC $\approx$ 0.604) are in same range as ProbLog (Lowd and Domingos 2007). ProPPR obtains an AUC of 0.797, which outperforms the prior results reported by ProbLog and MLN.

7 Related work

Although we have chosen here to compare mainly to MLNs (Richardson and Domingos 2006; Singla and Domingos 2006a), ProPPR represents a rather different philosophy toward language design: rather than beginning with a highly-expressive but intractable logical core, we begin with a limited logical inference scheme and add to it a minimal set of extensions that allow probabilistic reasoning, while maintaining stable, efficient inference and learning. While ProPPR is less expressive than MLNs (for instance, it is limited to definite clause theories) it is also much more efficient. This philosophy is similar to that illustrated by probabilistic similarity logic (PSL) (Brocheler et al. 2010); however, unlike ProPPR, PSL does not include a "local" grounding procedure. Our work also aligns with the lifted personalized PageRank (Ahmadi et al. 2011) algorithm, which can be easily incorporated as an alternative inference algorithm in our language.

Technically, ProPPR is most similar to stochastic logic programs (SLPs) (Cussens 2001). The key innovation is the integration of a restart into the random-walk process, which, as we have seen, leads to very different computational properties.

ProbLog (De Raedt et al. 2007), like ProPPR, also supports approximate inference, in a number of different variants. An extension to ProbLog also exists which uses decision theoretic analysis to determine when approximations are acceptable (Van den Broeck et al. 2010). Although this paper does present a very limited comparison with ProbLog on the WebKB problem (in Table 15) a further comparison of speed and utility of these different approaches to approximate inference is an important topic for future work.

There has also been some prior work on reducing the cost of grounding probabilistic logics: notably, Shavlik and Natarajan (2009) describe a preprocessing algorithm called FROG that uses various heuristics to greatly reduce grounding size and inference cost, and Niu et al. (2011) describe a more efficient bottom-up grounding procedure that uses an RDBMS. Other methods that reduce grounding cost and memory usage include "lifted" inference methods (e.g., Singla and Domingos 2008) and "lazy" inference methods (e.g., Singla and Domingos

Table 15 AUC results on the WebKb classification task. ProbLog results are from Gutmann et al. (2010), and MLN results are from Lowd and Domingos (2007)

	Co.	Wi.	Wa.	Te.	Avg.
ProbLog	–	–	–	–	0.606
MLN (VP)	–	–	–	–	0.605
MLN (CD)	–	–	–	–	0.604
MLN (CG)	–	–	–	–	0.730
ProPPR($\mathbf{w} = 1$)	0.501	0.495	0.501	0.505	0.500
ProPPR	**0.785**	**0.779**	**0.795**	**0.828**	**0.797**

The best results are highlighted in bold
Co. Cornell, *Wi.* Wisconsin, *Wa.* Washington, *Te.* Texas

2006b); in fact, the LazySAT inference scheme for Markov networks is broadly similar algorithmically to PageRank–Nibble–Prove, in that it incrementally extends a network in the course of theorem-proving. However, there is no theoretical analysis of the complexity of these methods, and experiments with FROG and LazySAT suggest that they still lead to groundings that grow with DB size, albeit more slowly.

As noted above, ProPPR is also closely related to the PRA, learning algorithm for link prediction (Lao and Cohen 2010). Like ProPPR, PRA uses random walk processes to define a distribution, rather than some other forms of logical inference, such as belief propagation. In this respect PRA and ProPPR appear to be unique among probabilistic learning methods; however, this distinction may not be as great as it first appears, as it is known there are close connections between personalized PageRank and traditional probabilistic inference schemes.[21] PRA is less expressive than ProPPR, as noted above. However, unlike PRA, we do not consider here the task of searching for paths, or logic program clauses.

8 Conclusions

We described a new probabilistic first-order language which is designed with the goal of highly efficient inference and rapid learning. ProPPR takes Prolog's SLD theorem-proving, extends it with a probabilistic proof procedure, and then limits this procedure further, by including a "restart" step which biases the system to short proofs. This means that ProPPR has a simple polynomial-time proof procedure, based on the well-studied personalized PageRank (PPR) method.

Following prior work on approximate PPR algorithms, we designed a local grounding procedure for ProPPR, based on local partitioning methods (Andersen et al. 2006, 2008), which leads to an inference scheme that is an order of magnitude faster than the conventional power-iteration approach to computing PPR, taking time $O(\frac{1}{\varepsilon\alpha'})$, independent of database size. This ability to "locally ground" a query also makes it possible to partition the weight learning task into many separate gradient computations, one for each training example, leading to a weight-learning method that can be easily parallelized. In our current implementation, an additional order-of-magnitude speedup in learning is made possible by parallelization. In addition to this, ProPPR's novel feature vector representation also extends SLP's semantics, and it is useful for learning to direct the proof search. Experimentally, we showed that ProPPR performs well on an entity resolution task, and a classification task. It also performs well on a difficult problem involving joint inference over an automatically-constructed KB, an approach that leads to improvements over learning each predicate separately. Most importantly, ProPPR scales well, taking only a few seconds on a conventional desktop machine to learn weights for a mutually recursive program with hundreds of clauses, which define scores of interrelated predicates, over a substantial KB containing one million entities.

In future work, we plan to explore additional applications of, and improvements to, ProPPR. One improvement would be to extend ProPPR to include "hard" logical predicates, an extension whose semantics have been fully developed for SLPs (Cussens 2001). Also, in the current learning process, the grounding for each query actually depends on the ProPPR model parameters. We can potentially get improvement by making the process of grounding more closely coupled with the process of parameter learning. Finally, we note that further speedups in multi-threading might be obtained by incorporating newly devel-

[21] For instance, it is known that personalized PageRank can be used to approximate belief propagation on certain graphs (Cohen 2010).

oped approaches to loosely synchronizing parameter updates for parallel machine learning methods (Ho et al. 2013).

Acknowledgments This work was sponsored in part by DARPA grant FA87501220342 to CMU and a Google Research Award. We thank Tom Mitchell and the anonymous reviewers for their helpful comments.

References

Ahmadi, B., Kersting, K., & Sanner, S. (2011). Multi-evidence lifted message passing, with application to pagerank and the kalman filter. In *Proceedings of the twenty-second international joint conference on artificial intelligence* (pp. 1152–1158).

Andersen, R., Chung, F. R. K., & Lang, K. J. (2006). Local graph partitioning using pagerank vectors. In *FOCS* (pp. 475–486).

Andersen, R., Chung, F. R. K., & Lang, K. J. (2008). Local partitioning for directed graphs using pagerank. *Internet Mathematics*, *5*(1), 3–22.

Backstrom, L., & Leskovec, J. (2011). Supervised random walks: Predicting and recommending links in social networks. In *Proceedings of the fourth ACM international conference on web search and data mining* (pp. 635–644).

Brocheler, M., Mihalkova, L., & Getoor, L. (2010). Probabilistic similarity logic. In Proceedings of the conference on uncertainty in artificial intelligence (pp. 73–82).

Callan, J., Hoy, M., Yoo, C., & Zhao, L. (2009). *Clueweb09 data set*.

Carlson, A., Betteridge, J., Kisiel, B., Settles, B., Hruschka Jr. E. R., & Mitchell, T. M. (2010). Toward an architecture for never-ending language learning. In *AAAI* (pp. 1306–1313).

Chakrabarti, S. (2007). Dynamic personalized PageRank in entity-relation graphs. In *Proceedings of the 16th international conference on world wide web* (pp. 571–580).

Cohen, W. W. (2000). Data integration using similarity joins and a word-based information representation language. *ACM Transactions on Information Systems*, *18*(3), 288–321.

Cohen, W. W. (2010). *Graph walks and graphical models*. CMU-ML-10-102, Carnegie Mellon University, School of Computer Science, Machine Learning Department.

Cussens, J. (2001). Parameter estimation in stochastic logic programs. *Machine Learning*, *44*(3), 245–271.

De Raedt, L., Kimmig, A., & Toivonen, H. (2007). Problog: A probabilistic prolog and its application in link discovery. In *Proceedings of the 20th international joint conference on artifical intelligence* (pp. 2462–2467).

Fuhr, N. (1995). Probabilistic datalog—A logic for powerful retrieval methods. In *Proceedings of the 18th annual international ACM SIGIR conference on research and development in information retrieval*. ACM (pp. 282–290).

Gutmann, B., Kimmig, A., Kersting, K., & De Raedt L. (2010). *Parameter estimation in problog from annotated queries*. CW reports (583).

Ho, Q., Cipar, J., Cui, H., Lee, S., Kim, J. K., Gibbons, P. B., et al. (2013). More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in neural information processing systems* (pp. 1223–1231).

Hoffmann, R., Zhang, C., Ling, X., Zettlemoyer, L. S., & Weld, D. S. (2011). *Knowledge-based weak supervision for information extraction of overlapping relations*. In *ACL* (pp. 541–550).

Jha, A., & Suciu, D. (2012). Probabilistic databases with markoviews. *Proceedings of the VLDB Endowment*, *5*(11), 1160–1171.

Lao, N., & Cohen, W. W. (2010). Relational retrieval using a combination of path-constrained random walks. *Machine Learning*, *81*(1), 53–67.

Lao, N., Mitchell, T. M., & Cohen, W. W. (2011) Random walk inference and learning in a large scale knowledge base. In *EMNLP*. ACL (pp. 529–539).

Lloyd, J. W. (1987). *Foundations of logic programming* (2nd ed.). Berlin: Springer.

Lowd, D., & Domingos, P. (2007). Efficient weight learning for markov logic networks. In *Knowledge discovery in databases: PKDD 2007*. Springer (pp. 200–211).

McCallum, A., Nigam, K., & Ungar, L. H. (2000). Efficient clustering of high-dimensional data sets with application to reference matching. In *Knowledge discovery and data mining* (pp. 169–178). http://dl.acm.org/citation.cfm?id=347123.

Muggleton, S. (1996). Stochastic logic programs. *Advances in Inductive Logic Programming*, *32*, 254–264.

Niu, F., Ré, C., Doan, A., & Shavlik, J. (2011). Tuffy: Scaling up statistical inference in markov logic networks using an RDBMS. *Proceedings of the VLDB Endowment*, *4*(6), 373–384.

Page, L., Brin, S., Motwani, R., & Winograd, T. (1998). The PageRank citation ranking: Bringing order to the web. Technical report, Computer Science department, Stanford University.
Poon, H., & Domingos, P. (2007). Joint inference in information extraction. In *AAAI* (Vol. 7, pp. 913–918).
Poon, H., & Domingos, P. (2008). Joint unsupervised coreference resolution with markov logic. In *Proceedings of the conference on empirical methods in natural language processing*. Association for Computational Linguistics (pp. 650–659).
Recht, B., Re, C., Wright, S., & Niu, F. (2011). Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems* (pp. 693–701).
Richardson, M., & Domingos, P. (2006). Markov logic networks. *Machine Learning*, *62*(1–2), 107–136. doi:10.1007/s10994-006-5833-1.
Shavlik, J., & Natarajan, S. (2009). Speeding up inference in markov logic networks by preprocessing to reduce the size of the resulting grounded network. In *Proceedings of the 21st international jont conference on artifical intelligence* (pp. 1951–1956).
Singhal, A. (2012). *Introducing the knowledge graph: Things, not strings*. The official Google blog: http://googleblog.blogspot.com/2012/05/introducing-knowledge-graph-things-not.html.
Singla, P., & Domingos, P. (2006a). Entity resolution with markov logic. In *Sixth international conference on data mining, 2006. ICDM'06* (pp. 572–582).
Singla, P., & Domingos, P. (2006b). Memory-efficient inference in relational domains. In *Proceedings of the national conference on artificial intelligence* (pp. 488–493).
Singla, P., & Domingos, P. (2008). Lifted first-order belief propagation. In *Proceedings of the 23rd national conference on artificial intelligence* (pp. 1094–1099).
Suchanek, F. M., Kasneci, G., & Weikum, G. (2007). Yago: A core of semantic knowledge. In *Proceedings of the 16th international conference on world wide web*. ACM (pp. 697–706).
Tong, H., Faloutsos, C., & Pan, J.-Y. (2006). Fast random walk with restart and its applications. In *ICDM*. IEEE Computer Society (pp. 613–622).
Van den Broeck, G., Thon, I., van Otterlo, M., & De Raedt L. (2010). Dtproblog: A decision-theoretic probabilistic prolog. In *AAAI* (pp. 1217–1222).
Wang, W. Y., Mazaitis, K., & Cohen, W. W. (2013). Programming with personalized pagerank: A locally groundable first-order probabilistic logic. In *Proceedings of the 22nd ACM international conference on information and knowledge management (CIKM 2013)* (pp. 2129–2138).
Zinkevich, M., Smola, A., & Langford, J. (2009). Slow learners are fast. *Advances in Neural Information Processing Systems*, *22*, 2331–2339.
Zinkevich, M., Weimer, M., Smola, A., & Li, L. (2010). Parallelized stochastic gradient descent. In *Advances in neural information processing systems* (pp. 2595–2603).